Dr. Dobb's Journal on CD-ROM

Database Management in C++

A single interface to multiple file formats

Art Sulger

Art specializes in database administration, analysis, and programming for the state of New York. He can be contacted on CompuServe at 71020,435.

Because individual database formats generally require that you write individualized code, programming to access database files can quickly become unnecessarily complex. In this article I present a class library which provides a single interface to multiple database file formats. In addition to freeing you from code duplication, this class structure allows your DBMS to support new data types such as those used with multimedia. (Imagine your xBase files holding sound and pictures!)

My original intent was to design a class structure for accessing xBase files. The structure had a parent *Database* class to provide portability; a *Table* class, in turn, descended from that. The *Table* class reads the xBase header and handled the opening and closing of files. The *Column* class implements the "table-has-columns" relation.

This design worked fine until I began to update a program based on the CData file format described in *C DataBase Development,* Second Edition, by Al Stevens (MIS Press, 1991). Instead of writing a new set of classes for this file structure, I developed the class structure in Figure 1, which moves duplicate code into two virtual classes, *Table* and *Column*. The advantage of this approach is that only 200 additional lines of code are necessary to provide support for a second database format. A user of the CData-based application also needs to access multiple related files. With the new design, it is easy to derive a "view" class from the *cDataTable* class and encapsulate the code that does the joins. And because a user of this class hierarchy sees any derived instance of *Table* as an instance of *Table*, the new design accommodates views constructed from different file formats.

The *LogicalDB* Class

At the top of the hierarchy is *LogicalDB* from which you derive *Table* and *Column*. *LogicalDB* is a convenient place to pool global resources used by all the descendants. *LogicalDB* gets its information from the system on which you run your application. When you compile it as a Windows app, for instance, it reads the INTERNATIONAL section of the WIN.INI file. If you compile it as a Presentation Manager app, *LogicalDB* gets the various *PM_National* settings. Or if you use neither of these platforms, *LogicalDB* reads and writes to its own default file.

The derived classes use an *enum SYSERR* type extensively. You can map these errors onto a STRINGTABLE in a GUI platform and build an error routine with platform-specific behavior. If you want to take advantage of platform-specific behavior, put that code into the *LogicalDB*. Listing One provides all of the headers, defined for OS/2, DOS, and Windows.

Because some operations work on sets of records, there is a NOTIFY_YESNO variable that you should set to "No" before doing Set operations to prevent informational messages from being repeated for every row. Most of these variables are static. Only one copy exists, no matter how many tables you open. A static counter (*ObjectID*) ensures that you initialize them once. The *Message()* member and its variables are not static, however. If you use these classes with a multitasking operating system, you may want to have different tables run in different threads, each with its own error mechanism.

The *LogicalDB* class instructs the *Column* object how to display numbers, dates, currency, and other system-defined data types; see Example 1. In this instance, the *Date* enum, which is in the *LogicalDB* class, is a switch variable for the *Column* class.

The *Table* Class

Table is an abstract class that descends directly from *LogicalDB*. A user of your derived classes will only use the public methods of the *Table* class. You should not add any methods to classes you derive from *Table* without first putting them into *Table*. The *Table* object is responsible for the storage of a single record. *Table* also maintains information about the current record pointer, the length of the record, and the name of the file that contains the record. You will have to override most of the methods in this class when you derive one for a specific DBMS, because different formats require different I/O.

The *Column* Class

The *Column* object is where the real action takes place. The *Column* object knows what type of data it is and knows how to display or "play" itself. I implemented only the basic types– NUMBER, CHARACTER, and CURRENCY. You must write your interpreters for the more exotic domains such as SOUND and COMMAND.

The *Table* object normally passes domain information into the *Column* object; otherwise, it will initialize a battery of *Column*s, doing the best it can to tell the *Column* object to which domain it belongs. For xBase files, this is easy because of the header information. An SQL derivation would query the SYSCOLUMNS tables for this information.

The real power of the *Column* class comes when you pass in domain information so that you can store a picture or sound in the file. Just write the picture displayer or sound interpreter, and you can "play" that structure as a native domain of your database. In the source code (provided electronically, see "Availability," page 3), for example, both CData and xBase files will have Timestamps and Record Sequence numbers, neither of which is native to the original format. The *Column* objects have two buffers: One is a pointer to the field's location in the record, and the other is a buffer to display the data.

Deriving a DBMS

Classes for specific formats descend from both *Table* and *Column*. xBase, the file layout used by Borland's dBase, Microsoft's FoxBase, Computer Associates' Clipper, and other systems, has extensive information about the file in a variable-length section at the beginning of the file. You can, for instance, find information about the row length, names, and types of the columns, and the date you last updated the file. In this respect, the format is similar to Paradox data files. I've provided *xBaseTable* and *xBaseColumn* classes in the source code.

The CData file format, however, has a dictionary bound into the application itself; therefore, no column-identifying information is available in the file. Consequently, you can send in the domain information from the application, or default everything to simple CHARACTER types. The classes *CDataTable* and *CDataColumn* illustrate this. If you don't send in domain information, *CDataTable* will read the first row to gather information about the lengths of the columns by counting the delimiting nulls. There is no way to deal with a CData file with no rows unless you explicitly send in domain information.

Again, the only truly public methods are those in the *Table* class. When you want to add another file format to your hierarchy, build the table and column class at this level, using the virtual methods as parents of your specific implementations. Be careful about adding new methods to your derivations that are too specific to the inherited class. You should be careful about adding new methods to your derived classes. To preserve the polymorphic nature of *Table*, all public methods, no matter how trivial, have to be part of the *Table* class.

Most of the *Column* methods are okay as is. *Column* has many methods that, although virtual, have instantiated code. *Column* assumes the DBMS stores dates CCYYMMDD. CData stores dates MMDDYY, so the *CDataColumn* class overrides the *AssignDate* and *DisplayDate* methods. CData also stores columns as ASCIIZ, that is, with a terminating null. The *CDataColumn* takes care of this by first calling the virtual parent's *Assign* method, then it puts the null in the proper location.

Further Developments

Because I plan to add a class for one of the SQL engines (such as Sybase or DB2/2), I designed the classes for expansion. (This will also make it possible for me to add an ODAPI interface in the future.) Consequently, as you move between different database implementations, you won't need to modify the application code (or retrain the application coders) if you use this design. Furthermore, these classes allow ordinary database formats to express themselves beyond their built-in character types. This is one way you can extend the life of those formats used in relational databases.

Aside from expanding them horizontally by including more database formats, you can grow the classes vertically. First, you derive classes to encapsulate methods for a specific table. For example, an *Employee* class would include, as members, the column structure and any business rules associated with an *Employee* object. You can overload the *Table* methods with *Employee*-specific code, then invoke the inherited method explicitly; see Example 2. You can also inherit classes from *Employee* and, for example, *Department*. This is how to create a "view"–a virtual table resulting from selecting, projecting, or joining rows from multiple tables. The tables can be in different formats on different hardware. Figure 2 shows a complete hierarchy (I've included only the first three levels in this article).

You may have noticed that there are no indexes in these classes because, at this generic level, I can't decide which format to support. Nor has speed been a problem because the code here executes quickly and my tables are small. For example, I have timed xFind at less than 4 seconds on a 5000 record file of 50-character records using a 486/66.

Figure 1 Initial class structure.

Example 1: The LogicalDB class is where the Column object learns how to display numbers, dates,

currency, and other system-defined data types.

```
static LogicalDB * db ;
switch ((int)db->Date)
 case (int)MMDDYY:
```

Example 2: Growing classes vertically.

```
void Employee::NewRow()
 {
 i = 0;
 while (key_array[i].tname != NULL)
  {
  key_array[i].IsModified = TRUE ;
  i++ ;
  }
 xBaseTable::NewRow() ;
 }
```

Example 3: This code will display a row to an output file.

```
for (i = 1; i < K.FieldCount() + 1; i++)
 write(out, bf, sprintf(bf, "%s ", K.Display(i)
)) ;
```

Example 4: This code prints all the rows in an xBase file where a column matches a value.

```
while (!K.IsEOF())
 {
 if (K.IsMatch(ColumnName, "Smith", exact))
  {
  for (i = 1; i < K.FieldCount() + 1; i++)
   cout << " - " << K.Display(i)
 ;
  cout << endl ;
  }
 K.Next() ;
 }
```

Example 5: Tables of this array can be xBase or CData.

```
xBaseTable & e = *new xBaseTable() ;
CDataTable & d = *new CDataTable() ;
struct Tab
 {
 Table & table ;
 } tab[] = {
  Tab(e)

 ,
  Tab(d)

 ,
 } ;
switch (iAction)
 case 0:tab[i].Close() ; break ;
```

Figure 2 Class hierarchy.

Listing One

```
// LogDB.hpp
//this encapsulates error messages
#ifndef LOGICALDB_HPP
#define LOGICALDB_HPP
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <iostream.h>
#include "System.h"
// errors are all positive so we can map into a Stringtable
// errors above 20,000 are fatal
enum SYSERR
 {
 GOOD_RETURN    = 0 ,
 // fatal system errors:
 D_OM          = 21000,   // out of memory
 D_INDXC        = 21001,   // index corrupted
 D_IOERR        = 21002,   // i/o error
 D_LOCK         = 21003,   // locking failure
 D_DEFAULTS     = 21004,   // corrupted or missing defaults
 // fatal dbms errors
 D_FORMAT       = 22000, // bad header info in data file
 D_PRIOR        = 22001,   // no prior record for this request
 D_FILENOTEXIST = 22002,
 D_MXTREESMAX   = 22003,
 D_BEYONDFILE   = 22004,
 D_DBNOTOPEN    = 22005,
 D_INDEXLOCKED  = 22006,
 D_DISKFULL     = 22007,
 D_OPENFAILED = 22008,
 D_CLOSEFAILED = 22009,
 D_READFAILED  = 22010,
 D_WRITEFAILED = 22011,
 D_CREATEFAILED = 22012,
 // dbms warnings:
 D_DUPL         = 2000,   // primary key already exists
 D_DEPEND       = 2001,   // dependent record exists
 D_NOPARENT     = 2002,   // no parent record exists for given key
 D_INDEXMAX     = 2003,   // index number > than max indices
 D_NOINDEXSET   = 2004,
 D_ACCESSDENIED = 2005,
 D_WAITCOUNT    = 2006,
 D_CASCADEFAIL  = 2007,   // Child Nullify or Delete failed
 D_NAMENOTFOUND = 2008,
 D_KEYISNULL = 2009,
 D_NOTUNIQUE = 2010,
 D_KEYPARTISNULL = 2011,
 // dbms notifications:
 D_NF          = 12003,   // record not found
 D_EOF          = 12004,   // end of file
 D_BOF          = 12005,   // beginning of file
 D_ZERORECS     = 12006,   // empty file
 D_NOTNEW       = 12007,   // New() not called before Write()
 D_NOTSELECT    = 12008,   // Cursor Selection require a SelectOpen()
 // Business Rules warnings:
 D_INVALIDDATE  = 1000,
 D_BADFORM      = 1001,    // error in sum, count or formula
 D_BADDOMAIN    = 1002
 };
#ifndef TRUE
#define TRUE    1
#endif
#ifndef FALSE
#define FALSE    0
#endif
#ifndef BOOL
```

```
#define BOOL      short
#endif
#ifndef UINT
#define UINT  unsigned int
#endif
#ifndef USHORT
#define USHORT  unsigned short int
#endif
#ifndef ULONG
#define ULONG  unsigned long int
#endif
enum NOTIFY_YESNO  // interupt extended operations for messages?
 {
 NO  = FALSE,
 YES = TRUE
 };
enum ARENULLS
 {
 NOTNULL, PARTLYNULL, ALLNULL
 };
const int MXKEYLEN     = 20 ;
const int MXCOLUMNWIDTH = 32 ;
const int MXCOLUMNNAME  = 32 ;
typedef enum enumCountry {
      OTHER=0,
      USA=1,
      CANADA=2,
      LATIN_AMERICA=3,
      NETHERLANDS=31,
      BELGIUM=32,
      FRENCH=33,
      SPAIN=34,
      ITALIAN=39,
      SWISS=41,
      DANISH=45,
      SWEDEN=46,
      NORWAY=47,
      GERMAN=49,
      AUSTRALIAN=61,
      JAPAN=81,
      KOREAN=82,
      SIMPL_CHINA=86,
      TRAD_CHINA=88,
      PORTUGUESE=351,
      FINNISH=358,
      ARABIC=785,
      HEBREW=972
      } eCountry  ;
typedef enum enumCurrrencyFormat
{CHARNUM,NUMCHAR,CHARSPACENUM,NUMSPACECHAR} eCurrencyFormat ;
typedef enum enumDate
{MMDDYY,DDMMYY,YYMMDD} eDate ;
typedef enum enumDigits
{ZERO,ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT} eDigits ;
//=============================================================
class LogicalDB
 {
 private :
  static short LogDBid ;  // construct this only once
  #if defined (PM_INCLUDED)
  HAB hab ;
  #elif defined (WINDOWS)
  HINST hInst ;
  #endif
 public :
  static BOOL ReadOnly ;
  static eCountry Country ;
  static eCurrencyFormat CurrencyFormat ;
  static eDate Date ;
  static eDigits Digits ;
```

```
        static char s1159[3], s2359[3], sCurrency[2],
                sThousand[2], sDecimal[2],
                sDate[2], sTime[2] ;

      char MessageBuffer[200] ;
      SYSERR er_num ;
      NOTIFY_YESNO notify ;
      SYSERR dberror(SYSERR e){er_num = e; dberror() ; return er_num ;}
      void dberror() ;
      int Message() ; // emit platform-specific messages
      LogicalDB(BOOL READONLY = TRUE) ;
      virtual ~LogicalDB() {if (LogDBid) LogDBid--;}
      void SetNotify(NOTIFY_YESNO x){notify = x ;}
      #if defined (PM_INCLUDED)
      void SetHab(HAB h){hab = h ;}
      #elif defined (WINDOWS)
      void SethInst(HINST h){ hInst = h ;}
      #endif
      SYSERR Error(){return er_num ; }
     } ;
    #endif // class LogicalDB

    #ifndef COLUMN_HPP
    #define COLUMN_HPP
    #include "LogDB.hpp"
    enum eElementType {
     CHARACTER     =0x0001,
     CURRENCY      =0x0002,
     DATE          =0x0004,
     DECIMAL       =0x0008,
     INTEGER       =0x0010,
     FLOAT         =0x0020,
     GRAPHIC       =0x0040,
     LOGICAL       =0x0080,
     MEMO          =0x0100,
     TIME          =0x0200, //hhmmss 24hr clock
     ZEROFILLED    =0x0400,
     SPACEFILLED   =0x0800,
     RSN           =0x1000,
     DOCUMENT      =0x2000,
     COMMAND       =0x4000,
     UPPER         =0x8000,
     LOWER         =0x00010000,
     CALCULATION   =0x00020000,
     WORDINTEXT    =0x00040000,
     NUMBER        =0x00080000,
     TIMESTAMP     =0x00100000,  //ccyy mm dd hh mm ss xxx
     DURATION      =0x00200000 , // hhh mm ss hx
     SOUND         =0x00400000
    };
    typedef struct COLUMNINFO
     {
     char *        FieldName ;
     eElementType  Type ;
     unsigned short Width ; // Unformatted storage width.
     unsigned short Decimals ;
     } COLUMN_INFO, * PCOLUMN_INFO ;
    enum ePrecision // For IsMatch().
     {
     exact,
     like
     } ;
    class Column : public LogicalDB
     {
     protected :
      char wrec[60] ;
      short bReadOnly ;
      eElementType ColType ;
      unsigned int rawWidth ;  // Does not include nulls for CData.
      char * cValue;           // raw value portion (created in Table)
```

```
        char * cDisplayValue ;
        char * cName ;          // External name of the Column.
        char * cDefault ;       // Filled in by NewRow()
        struct TabParms         // Table passes this in for RSN calc.
         {
         ULONG recordcount ;
         } * pTabParms ;
        unsigned int DispWidth;  // includes terminating null
        unsigned int nDecimals ; // Implied decimal for the stored value.
        int   rc ;              // Generic.
       public :
        Column() {cName=cDefault=cDisplayValue=NULL;   }
        virtual ~Column() ;
        virtual SYSERR      Assign(char * value) ;
        virtual SYSERR      AssignDate() ;
        virtual SYSERR      AssignNumber() ;
        virtual SYSERR      AssignTimeStamp() ;
        virtual SYSERR      AssignTime() ;
        virtual void        AssignDefault(void *) ;
        eElementType        ColumnType(){ return ColType ; }
        virtual const int   DayOf() ;
        virtual const char * Display() ;
        virtual const char * DisplayCurrency() ;
        virtual const char * DisplayDate() ;
        virtual const char * DisplayNumber() ;
        virtual const char * DisplayTime() ;
        const unsigned int  DisplayWidth(){return DispWidth - 1 ; }
        virtual void        Init   // allow Column arrays to be filled out
          (char        *Portion,
           char        *cName,    // column name
           eElementType  Type,    // see eElementType, above.
           unsigned int  ucLen,   // Storage length.
           unsigned short ucDec       = 0,
           char        *DefaultValue  =  NULL) ;
        BOOL               IsMatch(const char * Value, ePrecision p) ;
        virtual const char * Name(){return cName ; }
        void               SetDomain(eElementType x) ;
       } ;
      #endif // COLUMN_HPP

      #ifndef TABLE_HPP
      #define TABLE_HPP
      #include "LogDB.hpp"
      #include "Column.hpp"
      class Table: public LogicalDB
       {
       protected:
        Column  * * Col ;      // The attributes of the table
        int curr_fd;           // Current file descriptor
        BOOL bReadOnly, bSelect ;
        int i, j, rc ;         // Utility  vars.
        char * cRowBuffer ;    // Where the raw row is stored.
        enum eFileStatus
         {
         not_open,
         not_updated,
         updated
         } TabStat;
        struct TabParms        // Pass to Column in NewRow(0 processing.
         {
         ULONG recordcount ;
         } Tab_Parms ;
        ULONG ulRecordCount;
        ULONG ulCurrentRecord;
        ULONG ulRowSize ;
        BOOL E_O_F ;
        BOOL AlreadyRead ;
        BOOL IsNew ;
        unsigned int unFieldCount;
        unsigned int unCurrentFieldPointer ;
```

```
        char cFullFileName[128];
        void SetColumnDomain(int cl,eElementType d)//Make a Column all it can be!
         {Col[cl - 1]->SetDomain(d) ;}
       public :
        Table()
         {
         unCurrentFieldPointer=0;ulCurrentRecord=0;AlreadyRead=E_O_F=0;
         IsNew=0; bSelect = FALSE ;
         cRowBuffer=NULL;Col=NULL;
         }
        virtual ~Table() { ; }
        virtual SYSERR      Assign(int COL, char * data)
         {return Col[COL - 1]->Assign(data) ;}
        int             char2offCol(char * colname) ; // Name returns Number.
        virtual SYSERR      Close() = 0 ;
        virtual eElementType ColTypeXfrm(char hdr)
         {if (hdr == 'x') ; return CHARACTER ;}
        virtual char        ColTypeFromElement(eElementType e)
         {if (e == CHARACTER) ; return 'C' ;}
        const char        * ColumnName(int COL){ return Col[COL - 1]->Name() ; }
        virtual SYSERR      Create(char * fname, COLUMN_INFO c[]) = 0 ;
        virtual const int   DayOf(int COL)  // day part of date, timestamp
         {return Col[COL - 1]->DayOf() ; }
        virtual ULONG       Delete() = 0 ;
        virtual const char * Display(int COL){return Col[COL - 1]->Display() ;}
        virtual const UINT  DisplayWidths(int * ListOfColumns)
         {
         i = 0 ;
         while (*(ListOfColumns))
           i += Col[*(ListOfColumns)]->DisplayWidth() ;
         return i ;
         }
        virtual const UINT  DisplayWidth(int COL)
         {return Col[COL - 1]->DisplayWidth() ;}
        const unsigned int  FieldCount(){return unFieldCount ; }
        virtual const char * FirstColumnName()
         {
         unCurrentFieldPointer = 0 ;
         return Col[0]->Name() ;
         }
        BOOL            IsEOF() {return E_O_F ; }
        BOOL            IsColumn(unsigned int C)
         {
         if(C<1)return FALSE ;
         return (C>unFieldCount?FALSE:TRUE) ;
         }
        BOOL            IsColumn(char * C)
         {return (char2offCol(C)==-1?FALSE:TRUE) ;}
        BOOL            IsMatch(int ColNm, const char * Val, ePrecision e) ;
        BOOL            IsMatch
                      (const char * ColNm, const char * Val, ePrecision e) ;
        virtual const char * Name(){return cFullFileName ;}
        virtual SYSERR      Next() = 0 ;
        virtual const char * NextColumnName()
         {
         unCurrentFieldPointer++ ;
         if (unFieldCount <= unCurrentFieldPointer)
           return (char *)"" ;
         return Col[unCurrentFieldPointer]->Name() ;
         }
        virtual void        NewRow() = 0 ;
        virtual SYSERR      Open
        (char * name, BOOL readonly=FALSE, COLUMN_INFO c[]=NULL) = 0 ;
        virtual SYSERR      Top() = 0 ;
        eElementType        Type(int Cl){return Col[Cl-1]->ColumnType() ;}
        virtual SYSERR      Write() = 0 ;
      }; // end of class definition
      #endif // TABLE_HPP

      // xColumn.hpp
```

```
#ifndef XCOLUMN_HPP
#define XCOLUMN_HPP
#include "Column.hpp"
/*
 Native xBASE(c) columns are either LOGICAL, MEMO, NUMBER, CHARACTER
 or DATE.
*/
//=====================Column descendants================================
class xColumn : public Column
 {
 private:
 public:
  xColumn(){ ; }
  xColumn
   (char * PortionOfRowBuffer,
    char *        cName,    // column name
    eElementType   Type,    // LOGICAL, MEMO, NUMBER, CHARACTER, DATE
    unsigned short  rawSize,    // display (and storage) length
    unsigned short  decimals   = 0,
    char *        DValue  = NULL)
    {
    Init(PortionOfRowBuffer,
        cName, Type, rawSize, decimals, DValue);
    }
};
#endif

// xTable.hpp
#ifndef XTABLE_INC
#define XTABLE_INC
#include "Table.hpp"
#include "xColumn.hpp"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int const FIELD_REC_LEN = 32;  // length of field description record
int const HEADER_PROLOG = 32;  // Header without field desc and terminator
class xBaseTable: public Table
 {
 private :
  short iHeaderSize ;
  unsigned long ulFilesize ;
  struct     DBF
   {
   unsigned char    dbf_version;  // version character
   unsigned char    update_yr;    // date of last update - year(-1900)
   unsigned char    update_mo;    // date of last update - month
   unsigned char    update_day;   // date of last update - day
   ULONG           records;      // number of records in dbf
   unsigned short   header_length; // length of header structure
   unsigned short   record_length; // col lengths + 1 for delete mark
   unsigned char reserved_bytes[20] ;
   } dbf ;
  struct  FIELD_INFO          // This structure is filled in memory
   {                    // with a fread and passed to Column class
   char   name[11];          // name of field in asciz
   char   type;             // type of field...char,numeric etc.
   char field_data_address[4];// offset of field in record(not used here)
   unsigned char len;        // length of field
   unsigned char dec;        // decimals in field
   unsigned char reserved_bytes[14];  // reserved by dbase
   } header ;
  SYSERR Go(ULONG recno) ;
  BOOL   IsDeleted() ;
public:
  xBaseTable() {curr_fd = -1 ;}
  xBaseTable(char * FileName,  BOOL readonly=FALSE)
   {
   curr_fd = -1 ;
   Open(FileName, readonly) ;
```

```
        }
        ~xBaseTable(){Close() ; }
        SYSERR      Close() ;
        eElementType ColTypeXfrm(char header_type) ;
        char        ColTypeFromElement(eElementType e) ;
        SYSERR      Create(char * fname, COLUMN_INFO c[]) ;
        ULONG       Delete()
         {
         *(cRowBuffer)='*';
         if (Write()==GOOD_RETURN)return 1L ; else return 0L ;
         }
        SYSERR      Next() ;
        void        NewRow() ;
        SYSERR      Open(char *name, BOOL readonly=FALSE, COLUMN_INFO c[]=NULL) ;
        SYSERR      Top() ;
        SYSERR      Write() ;
      };
      #endif  // Table definitions

      // CDataColumn.hpp
      #ifndef CDATACOL_HPP
      #define CDATACOL_HPP
      #include "Column.hpp"
      /*
      CData columns are either Alphanumeric (CHARACTER),
                      Numeric(DECIMAL|ZEROFILLED,DECIMAL|SPACEFILLED),
                      DATE,
                      CURRENCY
      CData character columns are left-justified and space filled with a
      single terminating null.
      Numbers are right-justified and left filled with either spaces of zeros.
      Decimals are fixed.
      */
      //======================Column  descendants================================
      class CDataColumn : public Column
       {
       private:
        SYSERR      AssignDate() ;
        const char * DisplayDate() ;
       public:
        CDataColumn(){;}
        CDataColumn
         (char * PortionOfRowBuffer,
          char *        cName,    // column name
          eElementType    Type,
          unsigned short   rawSize,  // Does NOT include the Null!
          unsigned short   decimals   = 0,
          char *        DValue   = NULL)
          {
          Init(PortionOfRowBuffer,
              cName, Type, rawSize, decimals, DValue);
          }
        SYSERR       Assign(char * value)
          {
          *(cValue + rawWidth) = '\0' ;
          return Column::Assign(value) ;
          }
      }; // end of class definition
      #endif

      // CDatatab.hpp
      #ifndef CDATA_INC
      #define CDATA_INC
      #include "Table.hpp"
      #include "CDataCol.hpp"
      #include <stdio.h>
      #include <stdlib.h>
      #include <time.h>

      class CDataTable: public Table
```

```
    {
    private :
     struct    FHDR
      {
      ULONG      first_record ;
      ULONG      next_record ;
      short      record_length;
      } fhdr ;
     SYSERR Go(ULONG recno) ; // from One.
     BOOL IsDeleted() ;
    public:
     CDataTable() {curr_fd = -1 ;}
     CDataTable(char * FileName,  COLUMN_INFO col[], BOOL readonly=0)
      {
      curr_fd = -1 ;
      Open(FileName, readonly, col) ;
      }
     ~CDataTable(){Close() ; }
     SYSERR    Close() ;
     SYSERR    Create(char * fname, COLUMN_INFO c[]) ;
     ULONG     Delete() ;
     SYSERR    Next() ;
     void      NewRow() ;
     SYSERR    Open(char * name, BOOL readonly=FALSE, COLUMN_INFO c[]=NULL) ;
     SYSERR    Top() ;
     SYSERR    Write() ;
   }; // end of CDataTable
#endif  // Table definitions
//=========================System.h=========================
#ifndef SYSTEM_H
#define SYSTEM_H
#include <stdlib.h>
#include <direct.h>
#if defined __OS2__
#define INCL_WINSHELLDATA
#define INCL_WINDIALOGS
#define INCL_WINPOINTERS
#include <OS2.H>
#else
#ifndef BOOL
typedef  short BOOL ;
#endif
#endif
#if defined __OS2__ | __MSDOS__
#define PATH_SEPARATOR    "\\"
#else
#define PATH_SEPARATOR    "/"
#endif
#define  MAXPROFILEPATH   30
char * DataDirectory(char * szFullFileName)  ;
char * TimeStamp(char * bf) ;
BOOL IsBlank(const char * c, int wide) ;
BOOL IsValidDate(int iCCYY, int iMM, int iDD) ;
BOOL IsValidDate(char *CCYY, char *MM, char *DD) ;
int  Leapyear(int iYYYY) ;
LONG TimeHundreths(const char * v, char sep) ;
//-----------------------------------------------------
inline BOOL IsBlank(const char * c)
 {
 while ((*c == ' ') || (*c == '\t') || (*c == '\b') || (*c == '\n') || (*c == '\r')
 || (*c == '\n'))
   c++ ;
 return (*c == '\0') ;
 }
//-----------------------------------------------------
inline BOOL IsBlank(const char * c, int wide)
 {
 while ((*c == ' ')&&(wide > 0))
  {
  wide-- ;
```

```
        c++;
        }
      return (wide == 0);
      }
    #endif // SYSTEM_H
```